PyHTK: PYTHON LIBRARY AND ASR PIPELINES FOR HTK

C. Zhang, F.L. Kreyssig, Q. Li, P.C. Woodland

Cambridge University Engineering Dept., Trumpington St., Cambridge, CB2 1PZ U.K.

{cz277,flk24,ql264,pcw}@eng.cam.ac.uk

ABSTRACT

This paper describes PyHTK, which is a Python-based library and associated pipeline to facilitate the construction of large-scale complex automatic speech recognition (ASR) systems using the hidden Markov model toolkit (HTK). PyHTK can be used to generate sophisticated artificial neural network (ANN) models with versatile architectures by converting a compact configuration file defining the ANN, into the form used by HTK tools, as well as supporting a range of capabilities to train and test ANN models. The ASR pipeline is divided into multiple steps, which can be arranged and customised for different ASR data sets, and allows for both step-by-step and fully automatic end-to-end operation. PyHTK is integrated with HTK 3.5.1 which includes an expanded range of ANN layer types and very flexible ways to connect them, together with capabilities for ASR training and testing. Some example systems are included to illustrate the flexibility and performance achievable.

1. INTRODUCTION

HTK is an open-source research toolkit for automatic speech recognition (ASR), based on the C programming language, which has been continuously developed for almost three decades and has been available for free download since September 2000 [1]. During its long history, HTK has integrated many important hidden Markov model (HMM)-based techniques, such as phonetic decision trees for HMM state-tying [2], transform-based speaker adaptation [3], and lattice-based discriminative sequence training [4], into a common framework and has been widely used for both research and commercial ASR system development. HTK 3.5 [5, 6], was released in late 2015, which included native support for artificial neural network (ANN)-based acoustic models. However, in comparison to other ASR [7, 8] and deep learning toolkits [9–11], HTK 3.5 lacked ways to configure ANN models in a user-friendly fashion as well as a complete publicly available script pipeline that can perform large-scale ASR system construction automatically[1].

Reproducibility has long been an issue in ASR research, as practical ASR systems depend on complex pipelines and hyperparameters that are too detailed to fully list in a paper. The Kaldi toolkit [7] made a crucial contribution to reproducibility, by releasing alongside the source code for key speech processing functions, a collection of bash pipeline scripts for automatic ASR system construction, which are often specific and customised for each distinct speech data set. In mainstream deep learning toolkits, such as TensowFlow and PyTorch [10, 11], users can simply define model structures and training configurations via a simple Python script. This can then be shared for improved dissemination of the work.

This paper introduces PyHTK, a new Python-based function library and ASR pipeline for HTK, that has been developed at the Cambridge University Engineering Department (CUED) since 2015. Instead of wrapping the original C library with Python, PyHTK implements a set of new Python functions based on NumPy and SciPy, which covers many of the procedures used in ASR construction, and can interact with existing HTK tools. Experience has shown that the use of PyHTK greatly reduces the difficulty of both development and deployment of HTK. PyHTK also facilitates a major revision of the internal ASR pipeline scripts for HTK, some of which had been in use for many years and had been used for the construction of a wide range of HTK-based systems. A small set of Python scripts based on clearly defined training and test steps was created based on PyHTK. The pipeline can be customised for each particular task using task-specific configuration or config files, and can be used for ASR system construction in a step-by-step fashion or for a complete end-to-end build (from initial data to final system). Furthermore, HTK 3.5.1 has been developed with new support for more types of ANN layer, which includes support for long short-term memory (LSTM) [12] models, gated recurrent units (GRUs) [13], convolutional neural networks (CNNs) [14], a subsampling layer, a Gaussian mixture model (GMM) layer [15, 16], and a self-attentive layer [17]. In addition, the methods by which multiple vectors within an ANN are combined was extended. Previously only concatenation was supported for mixing features from multiple sources, whereas HTK 3.5.1 and PyHTK support additional mechanisms which in turn allow for a general use of residual connections, gating, and attention. Numerous capabilities for training and testing were also added to HTK, including various approaches to stochastic gradient descent (SGD), maximum likelihood (ML) sequence training, multi-channel raw waveform features [18], and lattice rescoring with LSTM-based language models [19].

The paper is organised as follows. Section 2 reviews the ANN model support in HTK 3.5, while the new features of HTK 3.5.1 are explained in Sec. 3. Section 4 introduces the design of the PyHTK library and Sec. 5 describes the ASR pipeline. Experiments on example systems are presented in Sec. 6, followed by conclusions.

2. GENERIC ANN SUPPORT IN HTK 3.5

HTK supports many HMM-based speech processing techniques covering the entire ASR pipeline. The integration of additional modules (C libraries) for ANN acoustic models in HTK 3.5 enabled the use of all previously established approaches in the design of ANN based ASR systems. The HTK tools for GMM-HMM systems, e.g. for alignment and decoding, were extended to support ANN-HMMs.

In a similar fashion to other deep learning and ASR toolkits, HTK 3.5 supports a range of ANN model structures that are equivalent to any directed cyclic graph by viewing each ANN layer as a

Many current & previous members of the CUED speech team have contributed to HTK. Mark Gales developed many aspects of a CUED-internal automatic ASR pipeline primarily based on shell scripts.

[1]Note that HTK has always included small-scale recipes and scripts that illustrate the main use of the tools.

graph node. In HTK 3.5 each ANN layer had to be a fully-connected (FC) layer, therefore only allowing deep feed-forward neural networks (DNNs), time-delay neural networks (TDNNs) and simple recurrent neural networks (RNNs). HTK implements this function via a structure called a *feature mixture* that is used to generate the input vector to a layer. A feature mixture can consist of any number of *feature elements*, where each of them is defined as the concatenation of vectors from the same *source* with different *context shifts*. The source of a feature element can be either input acoustic or augmented features, or the output values of an ANN layer. The term context shift refers to an integer indicating the time-step difference to the current time. For example, a TDNN can be implemented as a list of FC layers, where the feature mixture for each layer has a single feature element whose source is the directly preceding layer and the set of context shift values has multiple integers (e.g. {-1,0,1}). In order to use both CUDA and Intel MKL library functions to speed up matrix and vector operations, ANN layer parameters are organised as generic matrices and vectors, which can be used with both CPU and GPU support. These small parameter units allow for a light-weight speaker adaptation mechanism, which works by swapping speaker dependent parameters (matrices/vectors) according to speaker ids.

Error back propagation (EBP) with SGD is used as the standard optimisation approach in HTK 3.5 with many learning rate schedulers and gradient refinement techniques. Both frame level criteria, such as cross-entropy (CE) and minimum squared error, and lattice-based discriminative sequence criteria, such as maximum mutual information (MMI) and minimum phone error (MPE), are included. To implement these training criteria effectively, both frame and utterance level data shuffling approaches are needed. A multi-functional data cache was implemented for data shuffling with minimum I/O cost by pre-loading the data into the memory.

3. NEW FEATURES IN HTK 3.5.1

This section presents new features of HTK 3.5.1 which mainly belong to two categories: improved support for more complex deep ANN models and extra capabilities for training and testing.

3.1. Extended Feature Mixture and New ANN Layers

As reviewed in Section 2, given an ANN model with a static graph based architecture, HTK uses feature mixtures to define the composition of the input to each ANN layer as well as the connectivity among all layers. In HTK 3.5, only a concatenation operation was available to combine the component vectors from the feature elements to produce the mixed vector used as the input to the layer. In HTK 3.5.1, the feature mixture is extended to enable more types of operations for component vector combination. The currently supported operations are summarised below:

1. Concatenation: This is typically used for RNNs and TDNNs.
2. Addition: Element-wise addition of the component vectors, which can be used for residual connections.
3. Multiplication: Element-wised multiplication, which can be used for gating mechanisms.
4. Scaling: Scaling of each component vector by a separate factor, before any of the above operations. The factor can be derived from any dimension of any feature element source. This can be used for attention mechanisms.

In addition to the extended feature mixture that increases the choices for generating the input vectors to the layers, the choices of layer types are also increased. The previous release could only create DNNs, vanilla RNNs and TDNNs since only FC layers were supported. A full list of supported layers can be found in Table 1, which can be used as building blocks to construct more complex model architectures. A unique layer implemented in HTK 3.5.1 is the GMM layer, which is an alternative output layer that uses SGD to train a collection of GMMs [16]. It can therefore be used for joint training of a tandem system, which consists of GMM-HMM acoustic models and bottleneck features extracted by ANN models. Moreover, multiple activation functions are added, which include the recently proposed SELU [20] and Swish [21] activation functions.

Layer Type	Description
FC	Conventional fully connected layer
LSTM	LSTM model for one time step
GRU	GRU model for one time step
CNN	2D convolution with rectangle shaped filters
Subsampling	2D max pooling with rectangle shaped filters
GMM	Every output unit is a separate GMM
SelfAttentive	Self attention with modified penalty terms[17]
Permutation	Re-arrange the desired dimensions from input
ActivationOnly	FC layer with only the activation function
BiasOnly	FC layer with only a bias vector

Table 1. *A list of ANN layers supported by HTK 3.5.1. 2D stands for 2 dimensional.*

A number of example models built using early versions of HTK 3.5.1 are listed below:

1. DNN-GMM that has a GMM layer as the output layer, whose input are DNN extracted bottleneck features [16]. The GMMs can be modified through constrained maximum likelihood linear regression (CMLLR) based speaker adaptive training (SAT) [22].
2. Projected high order RNN (HORNNP) with stacked recurrent layers [23]. It was found to give similar WERs, whilst using half as many parameters as projected LSTMs (LSTMP) [24].
3. ResNet-TDNN is a TDNN whose hidden layers are replaced by a deep structure with a residual connection [25].
4. BD-FD-Grid-RNN-ResNet-TDNN uses a bi-directional (BD) and frequency-dependent (FD) grid-RNN as the input to the ResNet-TDNN [25].

3.2. ML Sequence Training and Two Model Re-Estimation

ML sequence training with EBP is implemented in HTK 3.5.1. Let λ be the composite HMM created using the reference labels of an utterance $\mathcal{O}$. For the objective function $\log p(\mathcal{O}|\lambda)$, we can obtain

$$\frac{\partial \log p(\mathcal{O}|\lambda)}{\partial \log a_k(t)} = \gamma_k(t) - y_k(t), \tag{1}$$

where $\gamma_k(t)$ is the posterior probability of HMM state k at time t given $\mathcal{O}$, and $a_k(t)$ and $y_k(t)$ are the input and output values of the softmax output activation function associated with target k. The value of $\gamma_k(t)$ can be calculated efficiently using the forward-backward algorithm (for HMMs) at the sequence level [6]. In HTK 3.5.1, this is easily implemented by reusing the existing code for GMM-HMM ML training. *Two model re-estimation* [6] is also available using an existing function, which uses a pre-trained model to generate $\gamma_k(t)$ in Eqn. (1) and can be viewed as a form of sequence level teacher-student training. Note that the ML training function in HTK can be configured for connectionist temporal classificiation [26] since it is equivalent to a special case of sequence level ML training with a special HMM topology [27].

3.3. Other New Training and Test Facilities

The capabilites of SGD training are enhanced in HTK 3.5.1. In addition to the previous absolute threshold update value clipping [6], a new relative threshold based clipping is provided [16], based on the distribution of gradients within a group of parameters. Another newly included method is the successive increase of the SGD mini-batch size according to a pre-specified frequency and step size, which was found helpful when training complicated models from scratch. Similarly, a momentum increase function was added to reduce the difficulty in using large momentum values. Nesterov momentum is also implemented as an alternative to the standard momentum method [28]. Other common SGD related functions, such as Batch Norm [29], Dropout [30], and the Adam learning rate scheduler [31], are included in HTK 3.5.1.

For testing more functionality is also added. For tandem systems, the dynamically generated bottleneck features can be expanded with differentials, normalised at different levels, and used to estimate feature transforms, such as CMLLR. This enables the reuse of tools and scripts designed for conventional GMM-HMMs to build and evaluate tandem systems. As mentioned in Section 3.1, HTK allows the GMMs and ANN bottleneck features of a tandem system to be jointly trained as an ANN model with a GMM output layer. For language models (LM), LSTM and GRU models are supported for lattice rescoring [19]. Low-frame-rate HMMs [32, 33], the CUDNN library, and multi-channel raw waveform input features [18] are supported for both training and testing.

4. PyHTK LIBRARY

As described in Sections 2 and 3.1, HTK can process ANNs with very flexible structures. However, defining a complex model structure in HTK is a problem. Previously, users had to write individual scripts to generate the actual HTK model file. Even though RNNs were supported in 3.5, they were very difficult to use due to needing to generate the model file with the unfolded RNN structure. The development of PyHTK significantly simplifies this procedure. Using PyHTK, any HTK model structure can be created using a config file. The design of the config file is based on the standard Python module `configparser`. An example config file defining a simple RNN with a single recurrent layer and sigmoid activation function is shown in Fig. 1. Beyond defining the ANN structure, the config file can also be used to define settings and hyperparameters for the training of the ANN structure. GMM-HMM models and training parameters can also be included in the config file.

Instead of interfacing Python directly with the HTK C library, PyHTK was designed to be standalone by re-implementing some important data structures from the HTK C library in Python. PyHTK can interact with HTK smoothly as it can read and write HTK model and data files. Since it is independently implemented from HTK, PyHTK has been developed using modern programming techniques such as object oriented programming and design patterns. In addition, PyHTK can dynamically unfold recurrent models, which allows recurrent layers to be presented to PyHTK as folded layers for efficient model definition (see Fig. 1), but as unfolded layers to HTK for effective frame level training with better data shuffling as

```
[ModelSet]
@FeatureType = <FBANK_D_Z>
InputObservation.Type = @FeatureType
@FeatureDim = 80
InputObservation.Dim = @FeatureDim
@RecurrentDim = 500
[NVector:ZeroVec]
Length = @RecurrentDim
Values = 0.0
[Layer:layer_rnn]
Kind = RNN
FeatureMixture.Num = 2
FeatureElement1.Dim = @FeatureDim
FeatureElement1.ContextShiftSet = {+5}
FeatureElement1.Source = @FeatureType
FeatureElement2.Dim = @RecurrentDim
FeatureElement2.ContextShiftSet = {0}
FeatureElement2.Source = ~V ZeroVec
UnfoldValue = 20
OutputDim = @RecurrentDim
ActivationFunction = Sigmoid
[Layer:layer_out]
Kind = FC
FeatureMixture.Num = 1
FeatureElement1.Dim = @RecurrentDim
FeatureElement1.ContextShiftSet = {0}
FeatureElement1.Source = layer_rnn
OutputDim = @auto
ActivationFunction = Softmax
[NeuralNetwork:RNN1L]
Layer2.Name = layer_rnn
Layer3.Name = layer_out
```

Fig. 1. *An example PyHTK config file that defines a sigmoid RNN with a single recurrent layer named as "RNN1L". Macro variables start with "@". "@auto" is a special macro whose value can be automatically set by PyHTK. "ZeroVec" is a zero-valued vector used as the initial values for the recurrent connection. "UnfoldValue" gives the number of steps for unfolding the recurrent layer.*

well as a proper truncation in EBP through time [34]. Any number of RNN layers is permitted, which are then unfolded for a desired number of time-steps. Given that the model parameters are stored in PyHTK using NumPy, parameters can be imported from a model trained using another toolkit that uses a Python-NumPy interface, such as PyTorch. Moreover, PyHTK is released and maintained via GitHub, and Sphinx Autodoc is adopted to produce detailed documentation.

5. PyHTK PIPELINE

Apart from the library functions for interaction with HTK, PyHTK also contains an ASR pipeline, which is adjustable according to different config files in order to apply it to different speech data sets.

This section first describes the key executable steps of the pipeline and then delineates a new framework for distributed SGD training that is easily obtained from the seamless integration of PyHTK and HTK.

5.1. Building Step Scripts for ASR Construction

The Python-based ASR pipeline is divided into multiple steps to modularise the major training and test procedures, and to allow for their re-arrangement for more flexible system construction. Each step contains a setup stage and a run stage. The setup stage does the preparation for the run stage, which includes creating directories and collecting resource files. The run stage executes the major jobs that are often computationally intensive. This two-stage design can reduce difficulties in revising and debugging the steps. All currently implemented steps are listed in Table 2, with more steps in preparation that will be released in future. The scripts can be used for a step-by-step or a complete end-to-end build (from initial data to final system). It is worth noting that whilst the step scripts are common to any speech data set, they can be configured using PyHTK's config files to meet task dependent requirements. More detailed descriptions, usage, and examples of the pipeline can be found in the HTKBook version 3.5.1.

Step	Description
mono	ML training for monophone GMM-HMMs
xwrd	Create cross-word triphone GMM-HMMs
xform	Linearly transform GMM-HMMs
rank	Redistribute Gaussian components in GMMs
sat	Speaker adaptive training with CMLLR
mpe	MPE/MMI training for GMM-HMMs
dnn-ce	CE training for ANN-HMMs
dnn-mpe	MPE/MMI training for ANN-HMMs
align	Generate alignments using Viterbi algorithm
latgen	Generate word or phone marked lattices
decode	Decode a set of HMM acoustic models
rescore	Rescore lattices with a different setup or LM
scoring	Score decoding results

Table 2. *The list of current building steps in the ASR pipeline.*

5.2. Framework for Distributed SGD Training of ANN models

Within the distributed training algorithm, each epoch is split into N *blocks*, and the algorithm splits the training data into N non-overlapping sub-sets. Each sub-set is used for one block. For M workers (processes/machines), each sub-set is divided evenly among the workers. Each worker independently updates a version of the model by training on its allocated subset using multiple SGD updates. Afterwards, the outcomes from the M workers are merged based on the method described below. Then the next block starts in the same fashion. This process is repeated until the epoch is finished.

The models updated by the workers are merged using PyHTK's functionality. The process is based on a binary tree through iterative merging of leaf-nodes. This process is illustrated in Fig. 2. First models from worker 1 and 2 as well as 3 and 4 are merged. This is done through a weighted average of their parameters (θ_1 and θ_2, θ_3 and θ_4 respectively). In order to reduce the sensitivity to combination weights, multiple options are compared, e.g. by testing on a validation set, and the best combination weights are chosen. The resulting two models are merged by a similar weighted average of their parameters θ_{12} and θ_{34}, resulting in the final parameters set θ_{out}. This merging procedure can improve on the block momentum method [35], which has been found to be sensitive to the hyper-parameters.

It is worth mentioning that alternative distributed training algorithms have also been developed at CUED [36].

6. EXPERIMENTS

To illustrate the range of model types supported by PyHTK, the models given in Section 3 were evaluated by training systems on multi-genre broadcast (MGB) data [37] from the MGB3 English speech recognition challenge task [38]. The experimental setup with a 63k word vocabulary is identical to those in [22, 23, 25]. The models that

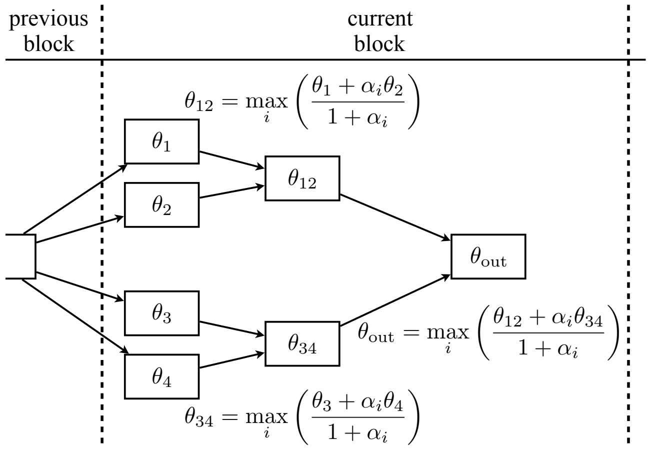
previous block
current block
$\theta_{12} = \max_i \left(\frac{\theta_1 + \alpha_i\theta_2}{1+\alpha_i}\right)$
θ_1
θ_2
θ_{12}
θ_{out}
θ_3
θ_4
θ_{34}
$\theta_{\text{out}} = \max_i \left(\frac{\theta_{12} + \alpha_i\theta_{34}}{1+\alpha_i}\right)$
$\theta_{34} = \max_i \left(\frac{\theta_3 + \alpha_i\theta_4}{1+\alpha_i}\right)$

Fig. 2. *Distributed SGD training using PyHTK. Pairs of models, trained on separate portions of data, are merged by weighted averaging. $\{\alpha_i\}$ is a set of pre-specified combination weights, and the resulted models are chosen based on validation set results.*

are compared are a 2-layer LSTMP baseline, a 2-layer HORNNP [23], the ResNet-TDNN and the BD-FD-Grid-RNN-ResNet-TDNN from [25], and a tandem system with a 7-layer DNN that was jointly optimised using the MPE training for DNN-GMM [22]. This last system uses HTK's special GMM layer and tandem system support, and can be improved using CMLLR based SAT. The MPE trained BD-FD-GridRNN-ResNet-TDNN results in a WER of 22.2%, which is not included in [25].

System	Criterion	vit	cn
LSTMP	CE	25.7	25.2
HORNNP	CE	25.6	25.2
ResNet-TDNN	CE	25.1	24.7
BD-FD-GridRNN-ResNet-TDNN	CE	24.6	24.3
DNN-GMM	MPE	26.0	25.7
DNN-GMM + CMLLR SAT	MPE	25.1	24.8
BD-FD-GridRNN-ResNet-TDNN	MPE	22.7	22.2

Table 3. *%WER for 275h systems on MGB dev17b with a trigram LM and Viterbi decoding (vit) or confusion network decoding (cn).*

7. CONCLUSIONS

We have presented PyHTK, which includes our recently developed Python library and the corresponding Python based ASR pipeline, as well as the new features in HTK 3.5.1 that PyHTK supports. PyHTK enables much easier use of the HTK tools, especially for the design and training of complex ANN architectures. It also contains a distributed SGD training framework using a special type of weighted model averaging. The ASR pipeline is carefully documented in HTKBook and can be customised to use for any speech recognition task. HTK 3.5.1 uses extended feature mixtures that cover many common operations seen in deep learning, and an extended selection of ANN layers is also supported. Maximum likelihood sequence training is enabled for both direct model training and teacher-student training. In future HTK and PyHTK releases, it is planned to include more deep learning models, 2nd-order optimisation methods, and lattice-free discriminative sequence training.

8. REFERENCES

[1] http://htk.eng.cam.ac.uk

[2] S.J. Young, J.J. Odell, & P.C. Woodland, "Tree-based state tying for high accuracy acoustic modelling," *Proc. Human Language Technology Workshop*, Morgan Kaufman, 1994.

[3] C.J. Leggetter & P.C. Woodland, "Maximum likelihood linear regression for speaker adaptation of continuous density hidden Markov models," *Computer Speech and Language*, vol. 9, no. 2, pp. 171–185, 1995.

[4] D. Povey & P.C. Woodland, "Minimum phone error and I-smoothing for improved discriminative training," *Proc. ICASSP*, Orlando, 2002.

[5] C. Zhang & P.C. Woodland, "A general artificial neural network extension for HTK", *Proc. Interspeech*, Dresden, 2015.

[6] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey, A. Ragni, V. Valtchev, P. Woodland, & C. Zhang, *The HTK Book (for HTK version 3.5)*, Cambridge University Engineering Department, 2015.

[7] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlíček, Y. Qian, P. Schwarz, J. Silovský, G. Stemmer, & K. Veselý, "The Kaldi speech recognition toolkit," *Proc. ASRU*, Waikoloa, 2011.

[8] P. Doetsch, A. Zeyer, P. Voigtlaender, I. Kulikov, R. Schlüter, & H. Ney, "RETURNN: The RWTH extensible training framework for universal recurrent neural networks," *Proc. ICASSP*, New Orleans, 2017.

[9] D. Yu, A. Eversole, M. Seltzer, K. Yao, Z. Huang, B. Guenter, O. Kuchaiev, Y. Zhang, F. Seide, H. Wang, J. Droppo, G. Zweig *et. al.*,"An introduction to computational networks and the computational network toolkit," Microsoft, Tech. Rep. MSR-TR-2014-112, 2014.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore *et. al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," *Proc. OSDI*, Savannah, 2016.

[11] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Alban, & A. Lerer, "Automatic differentiation in PyTorch," *Proc. NIPS AutoDiff Workshop*, Long Beach, 2017.

[12] S. Hochreiter & J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[13] J. Chung, C. Gulcehre, K.H. Cho, & Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv.org*, 1412.3555, 2014.

[14] Y. LeCun, L. Bottou, Y. Bengio, & P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[15] E. Variani, E. McDermott, & G. Heigold, "A Gaussian mixture model layer jointly optimized with discriminative features within a deep neural network architecture," *Proc. ICASSP*, Brisbane, 2015.

[16] C. Zhang & P.C. Woodland, "Joint optimisation of tandem systems using Gaussian mixture density neural network discriminative sequence training," *Proc. ICASSP*, New Orleans, 2017.

[17] G. Sun, C. Zhang, & P.C. Woodland, "Speaker diarisation using 2D self-attentive combination of embeddings," *Proc. ICASSP*, Brighton, 2019.

[18] Z. Tüske, P. Golik, R. Schlüter, & H. Ney, "Acoustic modeling with deep neural networks using raw time signal for LVCSR," *Proc. Interspeech*, Singapore, 2014.

[19] X. Liu, X. Chen, Y. Wang, M.J.F. Gales, & P.C. Woodland, "Two efficient lattice rescoring methods using recurrent neural network language models," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 34, no. 8, pp. 1438–1449, 2016.

[20] G. Klambauer, T. Unterthiner, A. Mayr, & S. Hochreiter, "Self-normalizing neural networks," *Proc. NIPS*, Long Beach, 2017.

[21] P. Ramachandran, B. Zoph, & Q.V. Le, "Searching for activation functions," *arXiv.org*, 1710.05941.

[22] Y. Wang, C. Zhang, M.J.F. Gales & P.C. Woodland, "Speaker adaptation and adaptive training for jointly optimised tandem systems," *Proc. Interspeech*, Hyderabad, 2018

[23] C. Zhang & P.C. Woodland, "High order recurrent neural networks for acoustic modelling," *Proc. ICASSP*, Calgary, 2018.

[24] H. Sak, A. Senior, & F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," *Proc. Interspeech*, Singapore, 2014.

[25] F.L. Kreyssig, C. Zhang, & P.C. Woodland, "Improved TDNNs using deep kernels and frequency dependent grid-RNNs," *Proc. ICASSP*, Calgary, 2018.

[26] A. Graves, S. Fernández, F. Gomez, & J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," *Proc. ICML*, Pittsburgh, 2006.

[27] H. Hadian, H. Sameti, D. Povey, & S. Khudanpur, "End-to-end speech recognition using lattice-free MMI," *Proc. Interspeech*, Hyderabad, 2018.

[28] I. Sutskever, J. Martens, G. Dahl, & G. Hinton, "On the importance of initialization and momentum in deep learning," *Proc. ICML*, 2013.

[29] S. Ioffe & C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv.org*, 1502.03167, 2015.

[30] N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, & R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[31] D.P. Kingma & J. Ba, "Adam: A method for stochastic optimization," *arXiv.org*, 1412.6980, 2014.

[32] D. Povey, P. Vijayaditya, D. Galves, P. Ghahremani, V. Manohar, X. Na, Y. Wang, & S. Khudanpur, "Purely sequence-trained neural networks for ASR based on lattice-free MMI," *Proc. Interspeech*, San Francisco, 2016.

[33] G. Pundak & T. Sainath, "Low frame rate neural network acoustic models," *Proc. Interspeech*, San Francisco, 2016.

[34] G. Saon, H. Soltau, A. Emami, & M. Picheny, "Unfolded recurrent neural networks for speech recognition," *Proc. Interspeech*, Singapore 2014.

[35] K. Chen & Q. Huo, "Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering," *Proc. ICASSP*, Shanghai, 2016.

[36] A. Haider & P.C. Woodland, "Combining natural gradient with Hessian free methods for sequence training," *Proc. Interspeech*, Hyderabad, 2018.

[37] P. Bell, M.J.F. Gales, T. Hain, J. Kilgour, X. Liu, P. Lanchantin, A. McParland, S. Renals, O. Saz, M. Wester, & P.C. Woodland, "The MGB challenge: Evaluating multi-genre broadcast media transcription," *Proc. ASRU*, Scottsdale, 2015.

[38] http://www.mgb-challenge.org